REPORT DOCUMENTATION PAGE

AFRL-SR-BL-TR-01-

0294

1. AGENCY USE ONLY (*Leave Blank*)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	2/28/01	Final (2/1/98 - 11/30/01)

4. TITLE AND SUBTITLE
A Formal Framework for Architectural and Integration Styles

5. FUNDING NUMBERS
F49620-98-1-0217

6. AUTHORS
Rose F. Gamble
Department of Mathematics and Computer Science

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
University of Tulsa
600 South College Avenue
Tulsa, OK 74104-3189

8. PERFORMING ORGANIZATION REPORT NUMBER
14-2-1203182

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Freeman A. Kilpartick, Capt. USAF NM
AFOSR/NM
801 N. Randolph St. Room 732
Arlington, VA 22203-1977

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for Public Release; Distribution is Unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Component interoperability has become an important concern as industry and government migrate legacy systems, integrate COTS products, and assemble modules from disparate sources into a single application. Interoperability problems, their complexity and their resolution, can mean the difference between the successful, seamless integration of software and the complete failure of a composite system. Emerging research has shown that interoperability problems can be traced to differences in the software architectures of the components and the integrated application environment. Current technology does not fully identify what must be made explicit about software architecture to aid in the prediction of interoperability problems. The overall goal of our research is to define and build this technology. We focus on three technical areas: analysis process, modeling, and case studies. The results of the research to date are the categorization of architecture characteristics, the definition of a set of common conflicts, and the initial development of a theory of interoperability.

14. SUBJECT TERMS
Architectural and Integration Styles

15. NUMBER OF PAGES
21

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-1
298-102

Final Report

February 28, 2001

Principal Investigator: Rosanne F. Gamble

Institution:
University of Tulsa
Department of Mathematical & Computer Sciences
600 South College Avenue
Tulsa, Oklahoma 74104

Grant Number: F49620-98-1-0217

Title of Research:
A Formal Framework for Architectural & Integration Styles

Abstract

Component interoperability has become an important concern as industry and government migrate legacy systems, integrate COTS products, and assemble modules from disparate sources into a single application. While middleware is available for this purpose, it often does not form a complete bridge between components and may be inflexible for the eventual evolution of the application. What is needed is explicit design information that will forecast a more accurate, evolvable, and less costly integration solution implementation. Emerging research has shown that interoperability problems can be traced to differences in the software architectures of the components and integrated application. Furthermore, the solutions generated for these problems are guided by an implicit understanding of software architecture. Current technology does not fully identify what must be made explicit about software architecture to aid in the comparison of architectures and the expectations of participating entities within an integrated application. Thus, there can be no relief in the expense or the duration of implementing long-term reliance on middleware.

The overall goal of our research is to define and build this technology. Toward this end there are many individual pieces that need to be distinguished and analyzed. We focus on three areas of technical progress: analysis process, modeling, and case studies. Our preliminary Integration Component Architecture Process (ICAP) has been refined. It now includes those architecture characteristics that we have defined as standard to architecture. Additionally, we have defined a set of common conflicts to begin the development of a theory of interoperability. We have preliminary connections from characteristic comparisons to predefined conflicts. We have maintained our stance on using the combination of UML [RAT99] and Object-Z [DKRS91] for semi-formal and formal modeling. We believe that the use of these languages will aid in the future research of connecting conflicts to integration elements. We continue to develop in-house case studies and are currently embarking on industry initiatives to further the prospect of technology transfer.

Objectives

One of the major industrial and governmental efforts in software development is to build or migrate applications using heterogeneous component systems. These component systems include legacy software, commercial-off-the-shelf (COTS) products, and software under design. This style of software development enjoys the benefits of reusability, adaptability, and evolvability, as applied to the individual components and the integrated system application. Therefore, it is important to facilitate the integration of the components such that the final application satisfies its requirements.

A major drawback of this software development approach is that integrating heterogeneous components can manifest difficult interoperability problems among component systems. These problems inhibit integration among components due to mismatches between data and control characteristics of their exposed interfaces. Integration solutions can be complex, hard to derive, and time-consuming to develop. Traceability from problem to solution is important because system upgrades may produce new interoperability problems, causing earlier solutions to become obsolete. This results in modifications to the original integration strategy. Thus, understanding the reasons for interoperability problems and being able to formulate methods of design for resolution is very valuable.

Commercial middleware products exist, but are applied after design decisions have been made. This makes it difficult to choose the right product, and often the final choice, when implemented, does not provide a complete, flexible and evolvable solution. Their usage can eliminate needed traceability back to the interoperability problems. In addition, with this implementation-based approach, there is no feasible way to determine if the middleware satisfies the stated, possibly critical, requirements of the integrated application. Therefore, while commercial products may result in a low cost, short-term solution, any subsequent changes to the integrated application can greatly increase the cost for its maintenance and upgrade.

The goal of this research is to facilitate integration by making architecture interoperability analysis part of application design. This is in contrast to current methodologies that have the developers choose, somewhat blindly, the application configuration and middleware and then architect around those choices. With our approach, integration problems are predicted and solutions are planned and evaluated prior to implementation. Since there is an implied understanding on the part of developers concerning the basic configuration and cooperation of the components in the application, interoperability analysis at the software architecture level is feasible.

The first year of funding was devoted to determining what was needed for analysis at the architecture level of abstraction. We developed a shallow understanding of the analysis process, as well as modeling approaches for architecture descriptions and integration elements. The objectives of the research reported for the second year of funding focused on deepening that understanding and bridging the gaps between the individual entities of architecture characteristics, conflicts, integration solutions, and middleware. The third year of funding was directed to more formal and theoretical aspects of process specification and interoperability

concerns. Through this effort, we have completed the initial assessment and development of an interoperability problem detection methodology within a unified framework as proposed.

Status of the Research Effort

1. Introduction

Interoperability problems are multi-dimensional and often require complex, compositional solutions. Although there are several strategies that currently exist for the integration of systems, many suffer from informality and are tightly coupled to particular domains and products. More importantly, interoperability problem prediction and integration solution design are only recently being addressed. In sharp contrast, the most common practice is to deploy a delayed, implementation-based approach to resolve interoperability problems, especially in cases where middleware products are used. The focus of our research is the early prediction and assessment of interoperability conflicts among interacting components and the generation of a verifiable integration infrastructure for resolving these conflicts.

Our research focuses on minimizing the detail needed for interoperability analysis when formulating consistent models across the various entities involved in composability assessment, e.g., software systems, application requirements, interoperability conflicts, and middleware. We use principles of software architecture as a basis for normalization. Specifically, we have identified architecture properties of component systems and the application requirements using empirical study to determine their influence on interoperability conflicts. In addition, we have discovered and modeled integration elements, i.e., low-level integration functions with various compositions that underlie middleware.

For the purposes of this research, we use the following terminology (see Figure 1). A module is a computational component, internal to a system, which interacts via connectors. A component is an independent system. The application is the integrated system of components.

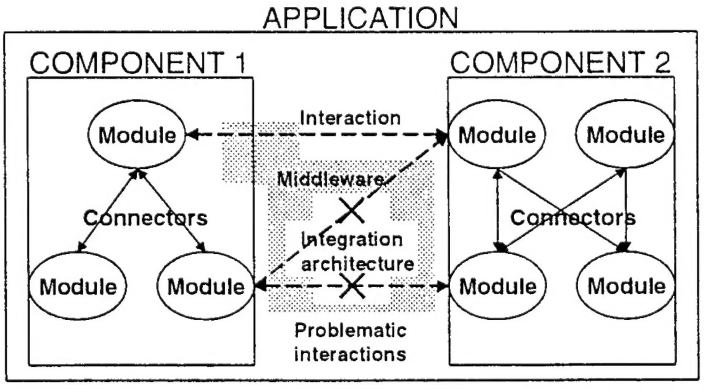


Figure 1: Terminology Usage

2. Technical Progress

The main objectives of the research as supported by AFOSR since February 1998 in the area of software architecture and interoperability are (1) to describe the non-functional properties of a

software system as being architecturally dependent, (2) to formally model the characteristics and domain components of software architectures, (3) to capture the formal underpinnings of interoperability through composition and integration of base architectural styles, (4) to prove properties of applications derived using formal models of architectures, and (5) to encompass the above formal modeling, integration, and property guarantees within a uniform formal framework. In this section, we summarize our current research over the past year toward achieving these objectives.

2.1 Software Architecture Characteristics for Interoperability.

Much research has been performed to describe a variety of software architecture characteristics [ABD96, AG97, SC97, GAO95, SIT97, KG99]. In [DGPK99], we established the validity of these characteristics toward achieving a complete and accessible set. We provided a comprehensive treatment of the various published characteristics (74 in all), using a combination of abstraction levels and semantic networks to show how they can be grouped for evaluation. The objective was to construct a representative set that includes characteristics embodying dominant, accessible component descriptions relevant to interoperability issues. We refer to this set as *architecture interaction characteristics*. These characteristics are partitioned according to two distinct perspectives of architecture description. The first perspective is from the *component-level*. These characteristics contribute to an understanding of the exposed interface of a participating component to other external subsystems. From the *application-level* perspective, characteristics formulate the architectural demands on configuration and coordination of the component systems into a single integrated application to satisfy requirements. In addition to the architecture interoperability analysis, we have evaluated the architecture interaction characteristics with respect to COTS product evolution within an integrated application [DPG00B].

CHARACTERISTICS	DEFINITION	VALUES
Data Storage Method	The details about how data is stored within a system [SIT97]	Repository, Data With Events, Local Data, Global Source, Hidden, and Distributed
Supported Data Transfer	The method supported by a particular architectural style to achieve data transfer [ABD96]	Explicit, Implicit, Shared
Data Topology	The geometric form the data flow takes in a system [SC97]	Hierarchical, Star, Arbitrary, Linear, and Fixed
Control Structure	The structure that governs the execution in the system [SIT97]	Single-Thread, Multi-Thread, Decentralized
Control Topology	The geometric form the control flow takes in a system [SC97]	Hierarchical, Star, Arbitrary, Linear, and Fixed
Identity of Components	Knowledge or awareness of other components in the system [SIT97]	Aware, Unaware
Blocking	Whether or not the thread of control is suspended [KG99]	Blocking, Non-Blocking

Table 1: Component-Level Characteristics

We analyzed the characteristics by eliminating redundancies among published characteristic definitions and partitioned them into three levels of abstraction: *orientation, latitude,* and *execution*. Each level designates at what point in the development effort the values of its

characteristics are established. Our goal was to determine the orientation level characteristics shown in Table 1 that represent high-level architectural requirements. Each of these characteristics is semantically related to at least one character in either the latitude or execution levels. From this solid foundation, we were able to strengthen our analysis efforts.

Table 2 defines our current set of application requirements at the orientation level. Some of the application-level characteristics are referred to by the same names as the component-level characteristics. However, they are defined by a different perspective or viewpoint. These characteristics are synthesized from the integrated application and environment requirements to formulate the architectural demands on configuration and coordination of the participating components in the application. For each characteristic within the above set, our previous work examined the definition through empirical studies to determine how its expectations for system performance across components cause interoperability conflicts [DPG00A]. This has led to our two-part approach to architecture interaction analysis (part of the pre-integration phase of ICAP [PKG99]) that we discuss in Section 2.4.

CHARACTERISTIC	DEFINITION	VALUES
Control Topology	The geometric formation of the control flow across the integrated application [SC97].	Hierarchical, Linear, Star, Arbitrary, Fixed
Data Topology	The geometric formation of the data flow across the integrated application [SC97].	Hierarchical, Linear, Star, Arbitrary, Fixed
Control Structure	The structure that governs the execution of the independent component systems in the integrated application [SIT97].	Single-Threaded, Multi-Threaded, Decentralized
Synchronization	Whether or not the components need to rendezvous [KG99, YBB99].	Synchronous, Asynchronous

Table 2: The Application-level Characteristics

2.2 An Initial Comparative Theory

We have begun preliminary research toward a theory of architecture interaction. As part of this theory, each component will have a set of values assigned to all known characteristics (Table 1). For each characteristic, there is at most one value, which will be assigned to it. The choice of value refers to the most restrictive possible for assignment. Analysis is first performed on a component-component basis by examining the following:

- Similar characteristics with like values
- Similar characteristics with mismatched values
- Different characteristics

By distinguishing these different assessments, we call attention to the fact that conflicts are not solely determined by comparing similar characteristics with mismatched values [ABD96,YBB99].

2.2.1 Categories of Conflict

We have found that repeated interoperability conflicts appear in one of three categories: transfer of control, transfer of data, and interaction initialization. For each of the categories, we have

described their conflict types using the same level of abstraction as the software architecture description [DPGK00].

Category 1: Control Transfer

1. Restricted points of control transfer
2. Unspecified control destination
3. Inhibited rendezvous
4. Sequencing multiple control transfers

Category 2: Data Transfer

5. Restricted points of data transfer
6. Unspecified data destination
7. Unspecified data location
8. Possible data inconsistency
9. Possible invalid data
10. Sequencing multiple data transfers
11. Mismatched data transfer assumptions

Category 3: Interaction Initialization

12. Initialization of control transfer
13. Initialization of data transfer

2.2.2 Notation

We define problematic architecture interactions as follows [DGPK00].

> **Definition:** A *problematic architecture interaction* is an interoperability conflict that is predicted through the comparison of architecture interaction characteristics and requires intervention via external services for its resolution.

The notation

$$x \rightarrow T \leftarrow y$$

means "x problematically interacts with y causing conflict set T" where T is a subset of common conflicts defined in [DGPK00] and x and y architecture interaction characteristics. Each problematic interaction maps to one or more of the above conflict types listed in section 2.2.1.

2.2.3 Assessment

The first step is to assess the details with respect to direct component interaction. The second step is to overlay the application requirements on the interacting components to refine the conflict set.

Step 1: Assessment Details

In this step, we first determine the values for the architecture interaction characteristics of each participating component. Using a bipartite graph, we perform pairwise assessment of components for all characteristics with values. Bipartite graphs are constructed for all component-component pairs, whether they will eventually communicate or not. The theory discussed in the previous section is used to map interactions to common conflict to determine which ones could be problematic. Union and additivity rules (part of the theory from [DGPK00]) are applied to clarify the set of problematic interactions into a minimal, understandable set in which characteristics and their conflict relationships are made apparent.

Step 2: Overlaying Application Requirements

The second step is to further refine the conflict set from step 1. We eliminate those problematic interactions from pairwise assessments in which there is no control and/or data exchange. Then, we append any new conflicts caused by the influenced of the application requirements. This is done by using the mapping supplied by the theory. However, the union and additivity rules both need to be extended with respect to application-level characteristics to prune and relate the conflicts from each perspective. This is part of our future efforts.

For illustration purposes, we present the two conflicts.

F1: Restricted points of control transfer
F2: Sequencing multiple control transfers

As an example, consider the bipartite graphs showing characteristics comparisons in Figure 2, given components *A*, *B*, and *C*. The dashed lines indicate there are no problematic interactions when directly comparing characteristics in *A* with characteristics in *B*. Assume that the application (*APP*) has a requirement that the control topology of the system of components is arbitrary. Empirical analysis shows that

$$CT.Hierarchical\ (A) \rightarrow \{\varnothing\} \leftarrow CT.Hierarchical\ (B)$$
$$CT.Hierarchical\ (B) \rightarrow \{F1,F2\} \leftarrow CT.Arbitrary\ (C)$$
$$CT.Hierarchical\ (B) \rightarrow \{F1,F2\} \leftarrow CS.Decentralized\ (C)$$
$$CT.Arbitrary\ (APP) \rightarrow \{F1, F2\} \leftarrow CT.Hierarchical\ (A), CT.Hierarchical\ (B)$$

where the *CT* and *CS* refer to the control topology and control structure, respectively. These relations indicate that both conflicts (*F1* and *F2*) result when *B* and *C* must interact. As a result of the *Union Rule for Problematic Interaction* [DGPK00], we can present the conflicts so that their relationship to each other is observable. In general, the union can indicate a single solution. Indeed, for this example, mediation between *B* and *C* can resolve both problems by determining an appropriate sequence of multiple control transfers and directing their point of entry.

The interaction assessment does not end with component-component interaction analysis. Using the characteristics from the application perspective, we can determine the potential for problematic interactions when components must satisfy the architecture requirements of the integrated system. Application requirements can influence interoperability by dictating a context in which certain configuration and coordination issues become problematic, depending on how compatible they are with the expectations of the components [DPG00A]. Thus, application-

component conflicts are variable, depending on the current application characteristics. In some instances, they can affect the resolution of a component-component conflict. Conversely, analysis of these requirements can render certain problematic interactions as irrelevant.

As shown in Figure 2, it appears that no component-component conflicts occur between *A* and *B*. However, when application requirements are examined, new problematic interactions with respect to *A* and *B* are discovered. In this case, the requirements for the application's control topology forces components which normally have a direct control exchange to communicate in a more decoupled manner. Now, intervention is needed in the form of external integration services, such as a mediator, to conduct the control transfer.

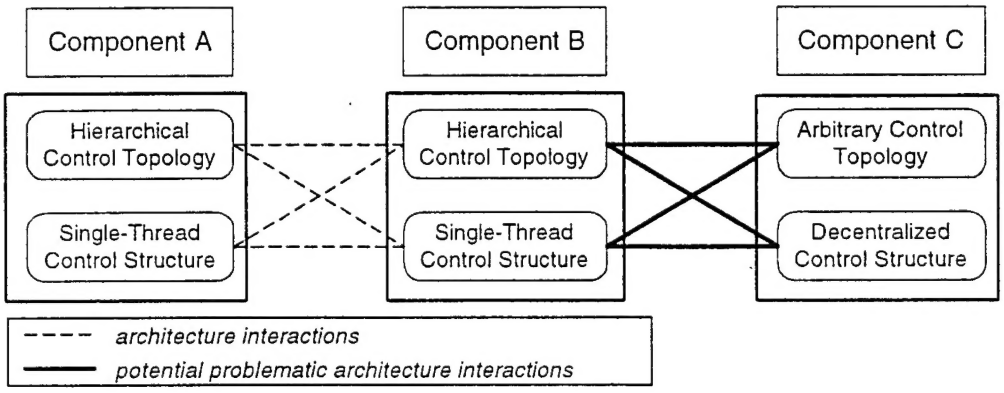


Figure 2: Architecture Interaction Types

2.3 Broadening the Characteristics to a Conspectus

Because architecture description for interoperability is a primary goal for this research, we clearly need a way to combine a minimal set of important properties for assessment into one place. We are experimenting with the use of an *architecture interaction conspectus* (AIC) that would be attached to each entity participating in an integrated system development effort. The conspectus forms a major building block for interoperability analysis by highlighting basic, yet relevant, software architecture properties, functional behaviors, and non-functional requirements. In this section, we briefly describe the context of the AIC and the type of assessment that can be performed.

2.3.1 Name and Type

Every entity that participates in the development of an integrated system has a name or identifier that separates it from other systems. We partition these entities into three types. The first type is the *application* that is composing independent subsystems. An application's indicators are in "goal" form, summarizing what is desired for the application as specified by its requirements. Thus, its indicators can be changed from their initial values as interoperability problems are discovered and corrected.

The second entity type is the *component* or participating independent subsystem. For the most part, components are complete, executable systems. Therefore, their indicators are stable.

However, some components may be in a design stage. In this case, components can have malleable "goal" indicators. Because applications can themselves be part of a larger complex system, they can also be considered components. With respect to the interoperability assessment that is performed using the AIC, we assume there is only one set of application requirements.

The third system type is *middleware*, i.e., those subsystems that provide integration services for the application. The manner in which middleware interfaces with the integrated environment (components, application requirements, and other middleware) indicates the need to ascertain distinct properties [MGR00].

Our investigation into the construction of the AIC focuses mainly on the application and its components as we present in the following sections. This starting point is facilitated by the similar properties depicting these two entities. Middleware, however, has some additional properties that need further research to accurately represent [MGR00]. However, we believe that it will share the same categories of properties presented next.

2.3.2 Style Related Characteristics

As discussed in Section 2.1, style-related characteristics have played a role in the type of interoperability problems that can be discovered by understanding the software architecture of a system. We union the set of characteristics in Tables 1 and 2 such that each characteristic is a separate indicator in the AIC. The interoperability assessment is performed using the process discussed in Section 2.3. The top portion of Figure 3 depicts the characteristics in the AIC.

2.3.3 Protocol Information

Because the style-related characteristics offer only a structural view of the component, behavioral characteristics are also needed for a more comprehensive analysis. As part of our effort to construct the AIC, we are working to provide meaningful indicators for protocol information that is defined in terms of the roles that a component may play in an application, and the protocols in which the roles participate. As a first pass, we divide protocol information into that which is available only by considering either component information or application information, forming two distinct indicators in the AIC (as seen in the middle portion of Figure 3). We do not detail the specific sequencing of messages, since this is beyond the scope of the AIC.

The protocol indicators are as follows.

> *Component Protocol Information.* This indicator specifies a list of the probable roles played by a particular component and the names of the protocols in which they participate. Although protocol names are present, there is no indication of how the roles participate in the protocol.
>
> *Application Protocol Information.* This indicator specifies the names of all protocols needed within an application. It also identifies the names of roles that should participate in each protocol. However, this role information is general and is not specific to any particular component.

These two indicators can provide an immediate view of expected behavior. With this information, it is possible to determine if the components that are present fulfill the behavioral expectations of the application. These indicators can also be used to decide if the addition of a component or use of an alternative component is feasible in terms of satisfying functional requirements. Moreover, the protocol information can be translated into an architecture definition language where expansion of the application and component specification can lead to deeper analysis. Hence, even a small amount of protocol information can be used to predict interoperability problems.

2.3.4 Non-Functional Property Expression

The style-related characteristics and the protocol behavior comprise very relevant information regarding interaction expectations. However, non-functional properties can also cause interoperability problems. Achieving non-functional requirements goals by looking only at individual components is a necessary first step to incorporate them into design decisions. Unfortunately, it is not well suited for analysis of heterogeneous component-based software systems, where the components must be viewed as a group with respect to application requirements. Our research examines the potential for delineating course-grained, non-functional properties which are the most pertinent when attempting an initial assessment or choosing components that better satisfy requirements to complement the information obtained by analyzing the architecture characteristics and protocols. Some of the more important extra-functional properties at the software architecture level ([BMRSS96, CNY95, SHA96]) are depicted in the lower portion of Figure 3.

- *Performance* refers to issues of time usage versus space needed by a system. Hence, the values **time** and **space** classify this property. These values can be ranked as either high or low.

- *Security* entails encryption strength, correctness, policies and protocols. Some quantifiable aspects of security are **encryption, authentication, mediation,** and **audit.** Due to the varying strengths, expenses and complexities of encryption available, it is ranked high, medium, low and none. All other values can simply be ranked high, low, none.

- *Modifiability* allows for evolution of **data constructs**, **functions**, and **objects** in a product. Due to the either/or nature of modifiability, a ranking of yes or no is assigned.

- *Reliability* delineates the soundness of **communications**, and the stability of **data** in an implementation. Assumptions about communications can be made according to the strategies of their transmissions. Thus, it is quantified by the presence of a direct or indirect scheme. Furthermore, data can be ranked as either volatile or persistent to denote its soundness in the application.

There are many other quantifiable aspects of each attribute. For example, access control mechanisms of the application could also be assessed with regard to security. Yet, experience allows the assumption that most software has some type of access control. Encryption, however,

is not typically present and, therefore, should be examined. Consequently, the values outlined in each category are necessary for a fundamental analysis.

It is important to note that the rankings of each value can be termed "fuzzy." For instance, should the time performance (with ranks low and high) of an application be medium to high, a high ranking will be chosen. Our assessment method specifies that if a weaker rank for a non-functional indicator is present in a collection of components, the rank for the application as a whole must abide by each lowest ranked property value, again loosely emulating the lattice structure inherent in the military security policy described in [PFL97]. For instance, should one component have non-modifiable functions, the system's functions as a whole would be considered non-modifiable.

2.4 Using an Architecture Interaction Conspectus

The characteristic indicators in the AIC are basic, yet powerful enough for values to be directly assigned and assessed for each component that participates in the integration, as well as the resulting application. If a value cannot be assigned, the indicator is simply not used in the assessment. There are two ways to address the limited number of conflicts that can be directly detected during analysis of partial specifications. First, because generic solutions will be designed to resolve the known conflicts, it is likely that they will also cover those that will not be discovered until the application design matures. Second, as the theory is expanded, values, as well as conflicts, may be derived from partial specifications.

Once established, it is a natural fit to express the indicators of a component's AIC within the eXtensible Markup Language (XML). This formulation is an XML *architecture interaction conspectus* (XMLAIC). Figure 3 shows the syntax of a portion of the conspectus as an XML Document Type Definition. In Figure 4, the XML documents depict sample XMLAICs according to the example presented in section 2.2.2 (see Figure 2).

The XMLAIC offers options for the pre- and post-purchase assessment of components. It also supports a mobile and evolvable means to perform interoperability analysis. The XMLAIC in the form of a XML document can be offered by a vendor on their web site for potential customers to assess the applicability of their product in any integration. Through a thin client, the vendor-provided XMLAIC can be used as input to determine potential problematic interactions at the level of software architecture. Because the XMLAIC is posted and maintained by the vendor, versioning of a product will be reflected so that customers can judge the impact those changes may have on existing integrated application. Also, the XMLAIC can be archived by the customer through its packaging with the currently used product. In this way, integration efforts can be compared, and the negative impacts of product evolution on an integrated application can be discovered prior to upgrade [DPG00c].

```
<!-- Element declarations -->
<!ELEMENT Conspectus (CharacterAnalysis?)>
<!ATTLIST Conspectus
        name CDATA #REQUIRED
        type (application | component | middleware) #REQUIRED>

<!-- Characteristic Section -->
<!ELEMENT CharacterAnalysis (ControlTopology?, ControlStructure?)>
<!ELEMENT ControlTopology EMPTY>
<!ATTLIST ControlTopology
        value (hierarchical | star | arbitrary | linear | fixed) #REQUIRED>
<!ELEMENT ControlStructure EMPTY>
<!ATTLIST ControlStructure
        value (single-thread | multi-thread | decentralized) #REQUIRED>
```

Figure 3: A Portion of the XMLAIC Definition

```
<Conspectus name="A" type="component">
  <CharacterAnalysis>
    <ControlTopology value="hierarchical"/>
    <ControlStructure value="single-thread"/>
  </CharacterAnalysis>
</Conspectus>
```

```
<Conspectus name="B" type="component">
  <CharacterAnalysis>
    <ControlTopology value="hierarchical"/>
    <ControlStructure value="single-thread"/>
  </CharacterAnalysis>
</Conspectus>
```

```
<Conspectus name="C" type="component">
  <CharacterAnalysis>
    <ControlTopology value="arbitrary"/>
    <ControlStructure value="decentralized"/>
  </CharacterAnalysis>
</Conspectus>
```

```
<Conspectus name="APP" type="application">
  <CharacterAnalysis>
    <ControlTopology value="arbitrary"/>
  </CharacterAnalysis>
</Conspectus>
```

Figure 4: Example XMLAICs

2.5 Interoperability and the Integration Elements

Though middleware frameworks are a popular solution to interoperability problems, they are often implemented in an ad hoc fashion. With this type of development, it is difficult, if not impossible, to show that the chosen middleware implementation satisfies integration requirements. As a result, our research continues to focus on realizing a formal, yet useable foundation to generate integration requirements and to validate an integration solution.

In earlier research, we defined an *integration architecture* to be the software architecture description of a solution to interoperability problems between at least two interacting component systems, and established that an integration architecture underlies each common middleware framework [KES99]. As a result, we describe integration architectures as compositions of *integration elements*. Integration elements are defined as one of the three high-level architecture connector patterns: *translator*, *controller*, and *extender* [KES99]. A translator converts data and functions between component system formats and performs semantic conversions. A controller coordinates and mediates the movement of information between component systems using predefined decision-making processes. An extender adds new features and functionality to one or more component systems to adapt behavior for integration. Previously, we employed a taxonomy

[KG98, KES99] to demonstrate the relationship of the integration elements to design patterns [GHJV95] that resolve interoperability conflicts and middleware frameworks.

Currently, our approach to composing integration elements focuses on studying a set of published middleware frameworks and their solutions to our set of common interoperability problems. However, developers from industry report that a single middleware product is not necessarily the answer to a heterogeneous conflict set, meaning that there are cases when pieces of different products are used as an integration solution. Our response to this observation is to separate the concerns of determining the integration solution requirements from determining which middleware framework may embody the solution. In this respect, we plan to focus on building more generic integration architectures and showing how they satisfy those requirements without slanting the architecture toward a known framework. The approach would allow developers to select functionality present in one framework and combine it with functionality from other frameworks, possibly resulting in customized, solutions without the overhead of an expert consultant. Using the taxonomy mentioned earlier, it is also possible to determine if a refinement of a framework produces a desirable solution. Furthermore, as integration efforts mature, discovery of new frameworks could be facilitated by the use of a generic integration architecture as part of the design effort.

Consider as an example, the generic shared repository integration architecture in Figure 4. Assume that multiple interacting components conflict with respect to data exchange because they have different data representations and they use distinct repositories to obtain data (which becomes redundant across the entire application). A potential integration solution would include a shared repository. Uniform modeling of the shared repository provides an understanding of the underlying integration functions and their composition to form a complete solution. Furthermore, it provides a basis from which to construct a formal model.

Translators address the different data formats of the components. We separate what would likely be implemented as a bi-directional translator to and from each component into two uni-directional translators that embody a single mathematical function for modeling ease. Thus, every component is associated with an INTranslator and an OUTranslator. The Sequencer (a variant of the controller integration element) receives input from multiple translators to determine the request sequence passed to the database. The database is an instance of an extender that represents the merging of the component stores. A Determiner (a variant of the controller integration element) passes the results of the database by deciding among the OUTranslators which component receives which result.

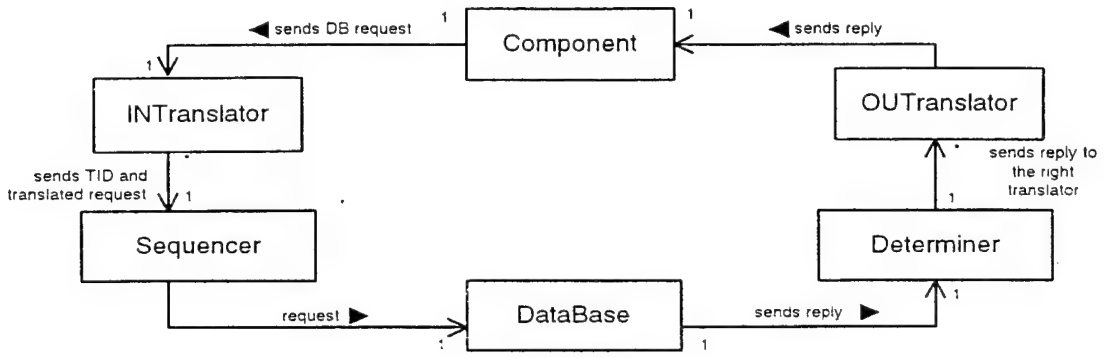


Figure 4: Generic Shared Repository Integration Architecture

To refine the generic architecture, suppose that Oracle is used to implement the database. What types of middleware frameworks possess the necessary control and translation function to work with the database implementation? It is possible to compare various frameworks to discover which best represents and can implement this functionality, or we can use the aforementioned taxonomy to find a pattern-based solution. For instance, the Mediator [GHJV95] design pattern is a refinement of the composition of the translators and controllers in Figure 4 and can be used to implement what we have shown above (and modeled formally in [PGKD00]). This refinement is shown in Figure 5 below.

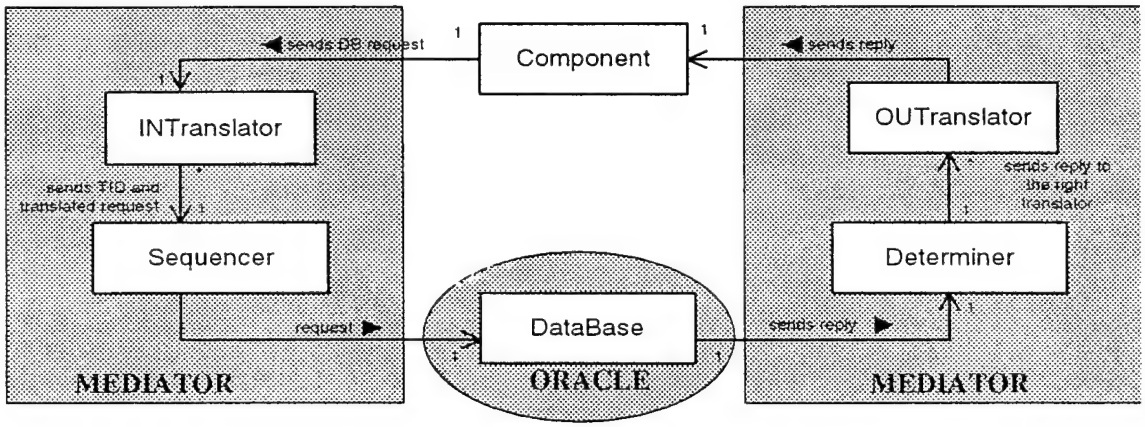


Figure 5: Specific Shared Repository Integration Architecture

The next step requires us to classify the internal parts of middleware frameworks to make a match of one or more that implements the desired mediators, without adding extraneous functionality.

3. Potential Transitions

We are still working with personnel from the Williams Companies, a Tulsa-based company, to conduct experiments using our technology. This project we are engaged in refers to the integration of software components from different organizations that are all part of a new online trading system for oil and gas commodities. Another Tulsa-based company, WorldCom

(formerly MCI WorldCom) has allowed us to experiment with an integration problem an involving e-commerce application and its integration with proprietary WorldCom systems that need to deliver dynamic, event-based information to a thin client. In addition, we are beginning to interface with Chevron with the intent of analyzing their software systems and those of Texaco for interoperability analysis in order to facilitate an information merger between the companies. We continue to look for applications in the Air Force and other government agencies where our technology may be applicable.

4.References

[ABD96] A. Abd-Allah, Composing heterogeneous software architectures, Ph.D. Diss., Dept. of CS, USC, Aug 1996.

[AG97] R. Allen & D. Garlan, A formal basis for architectural connection, *ACM TOSEM*, 1997.

[BMRSS96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad & M. Stal, *Pattern-Oriented Software Architecture: A System of Patterns,* John Wiley & Sons Ltd, 1996.

[CNY95] L. Chung, B. Nixon, & E. Yu, Using non-functional requirements to systematically select among alternatives in architectural design. In the *1sr International Workshop on Architectures for Software Systems,* 1995.

[DKRS91] R. Duke, P. King, G. Rose, & G. Smith, The Object-Z specification language, Software Verification Research Centre, Department of Computer Science, The Univ. of Queensland, TR 91-1.

[DGPK99] L. Davis, R. Gamble, J. Payton, & A. Kelkar, The impact of component architectures on interoperability, submitted to *Journal of Systems and Software*, under review, 1999.

[DPG00A] L. David, J. Payton, & R. Gamble, How system architectures impeded interoperability, to appear in the *2nd International Workshop on Software and Performance,* 2000.

[DPG00B] L. Davis, J. Payton, & R. Gamble, Toward identifying the impact of COTS evolution on integrated systems, In the *Workshop for Continuing Collaborations for Successful COTS Development*, May 2000.

[DPGF00] L. Davis, J. Payton, R. Gamble, & D. Flagg, Component indicators for architectural interaction assessment, submitted to *ICSE-2001*, August 2000.

[DGPK00] L. Davis, R. Gamble, J. Payton, & A. Kelkar, From feature interaction to architecture interaction: adapting processes for conflict detection, submitted to *ICSE-2001*, August 2000.

[GAO95] D. Garlan, R. Allen, & J. Ockerbloom, Architectural mismatch or why it is so hard to build systems out of existing parts, *Proceedings of the 17th International Conference on Software Engineering*, April 1995.

[GHJV95] E. Gamma, R. Helm, R. Johnson, & J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software,* Addison-Wesley, 1995.

[KG99] A. Kelkar and R. Gamble, Understanding the architectural characteristics behind middleware choices, *Proc. 1st Conf. on Software Reuse and Integration,* Sept. 1999.

[KG98] R. Keshav & R. Gamble, Towards a taxonomy of architecture integration strategies, *3rd Int'l Software Architecture Workshop,* Nov. 1998.

[KES99] R. Keshav, Architecture integration elements: Connectors that form middleware, M.S. Thesis, Dept. Math&CS, Univ. of Tulsa, June 1999.

[KK98] D. Keck & P. Kuehn, The feature and service interaction problem in telecommunications systems: A survey, *IEEE Transactions on Software Engineering,* 24(`0):779-796, October 1998.

[PGKD00] J. Payton, R. Gamble, S. Kimsen, & L. Davis, The opportunity for formal models of integration, In *the 2nd International Conference on Information Reuse and Integration,* forthcoming Nov. 2000.

[PKG99] J. Payton, R. Keshav & R. Gamble, System development using the integration component architecture process, *ICSE-99 Workshop on Successful COTS Development,* May 1999.

[PFL97] C. Pfleeger, *Security in Computing - Revised Edition,* Prentice-Hall, 1997.

[RAT99] http://www.rational.com, 1999.

[sha96] M. Shaw, Truth vs. knowledge: The difference between what a component does and what we know it does, In the *8th International Workshop on Software Specification and Design,* 1996.

[SG97] M. Shaw & P. Clements. A field guide to boxology: Preliminary classification of architectural styles for software systems. Proc. COMPSAC97 and *1st Int'l Computer Software & Applications Conf.*, Aug. 1997.

[SG96] M. Shaw & D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline,* Prentice Hall, 1996.

[SIT97] R. Sitaraman, Integration of software systems at an abstract architectural level, M.S. Thesis, Dept. Math&CS, Univ. of Tulsa, 1997.

[STI97] P. Stiger, An assessment of architectural styles and integration components, M.S. Thesis, Dept. Math&CS, Univ. of Tulsa, Dec. 1997.

[YBB99] D. Yakimovich, D. Bieman, & V. Basili, Software architecture classification for estimating the cost of COTS integration, *21st International Conference on Software Engineering,* 296-302, 1999.

Publications

Accepted

L. Davis, R. Gamble, & J. Payton. The impact of component architectures on interoperability, to appear in *Journal of Systems and Software,* 2002.

J. Payton, L. Davis, D. Underwood, and R. Gamble, Using XML for an architecture interaction conspectus, *XML Technology and Software Engineering, (XSE 2001),* May 2001.

J. Payton, R. Gamble, S. Kimsen, & L. Davis, The Opportunity for Formal Models of Integration. In the *2nd International Conference on Information Reuse and Integration,* Nov. 2000.

L. Davis, J. Payton, & R. Gamble, How System Architectures Impeded Interoperability, *2nd International Workshop on Software and Performance,* 2000.

N. Medividovic, R. Gamble, & D. Rosenblum., Towards Software Multioperability: Bridging Heterogeneous Software Interoperability Platforms, *4th International Software Architecture Workshop,* February 2000.

L. Davis, J. Payton, & R. Gamble. Toward Identifying the Impact of COTS Evolution on Integrated Systems, In the *Workshop for Continuing Collaborations for Successful COTS Development*, May 2000.

T. Shaft & R. Gamble, A theoretical basis for the assessment of rule-based system reliability, *Foundations of Information Systems: Toward a Philosophy of Information Systems,* 2000.

A. Kelkar & R. Gamble, Understanding the architectural characteristics behind middleware choices, *1st Int'l Conference on Information Reuse and Integration,* Oct. 1999.

J. Payton, R. Keshav, & R. Gamble, System development using integrating component architectures process, *Ensuring Successful COTS Development Workshop,* associated with *ICSE-99*, May 1999.

R. Gamble, P. Stiger, & R. Plant, Rule-based systems formalized within an architectural style, *Knowledge-based Systems Journal*, 1999.

K. Hasler, R. Gamble, K. Frasier, & T. Stiger, The potential for formally modeling architectural style abstractions within a taxonomy, position paper, *1st Working IFIP Conference on Software Architecture*, Feb. 1999.

R. Keshav & R. Gamble, Towards a taxonomy of architecture integration strategies, *3rd International Software Architecture Workshop*, Orlando FL, Nov. 1998.

R. Gamble & K. Frasier, Analyzing integration from an architectural perspective, *OOPLSA '98 Workshop on Applying Software Architecture as a Method*, Oct. 1998

Submitted

L. Davis, R. Gamble, J. Payton, G. Jonsdittur, D. Underwood, A notation for problematic architecture interactions, submitted to *Foundations of Software Engineering,* March 2001.

In preparation

S. Kimsen, L. Davis, & R. Gamble, Extending KBSs to incorporate external knowledge sources, in preparation for submission to *IEEE Transactions on Knowledge and Data Engineering, 2001.*

L. Davis and R. Gamble, Using ICAP in the classroom: experiments in integration analysis, in preparation for submission to *SIGCSE 2002.*

J. Payton, R. Gamble, & L. Davis, Developing formal models of integration using UML and Object-Z, in preparation for submission to the *International Journal of Computers and Their Applications*, 2001.

Personnel Associated with Research Effort

Faculty

Rosanne F. Gamble, Principal Investigator

Graduate Students

Reshma Keshav, M.S. June 1999, Master's Thesis title: *Architecture Integration Elements: Connector Models that Form Middleware.*
Kandi Frasier, M S. December 1999, Master's Report title: *Analyzing Middleware Frameworks.*
Sonali Kimsen, M.S. June 2000, Master's Report title: *An Integration Pattern for Knowledge Based Systems.*
Anand Kelkar, M.S. December 2000, Master's Report title: *Case Studies for Architectural Conflicts*
Leigh Davis, M.S. May 2001, Master's Thesis title: *Examining Software Architecture Property Interactions and the Influence of System Requirements*
Gerdur Jonsdottir
Charles Underwood

Undergraduate Students

Jamie Payton
Daniel Flagg

Interactions

Participation and presentations

Leigh Davis attended ICSE 2000 in Ireland, and participated and represented our papers in the 2^{nd} *Ensuring Successful COTS Development Workshop* and the 4^{th} *International Workshop on Software Architecture.*

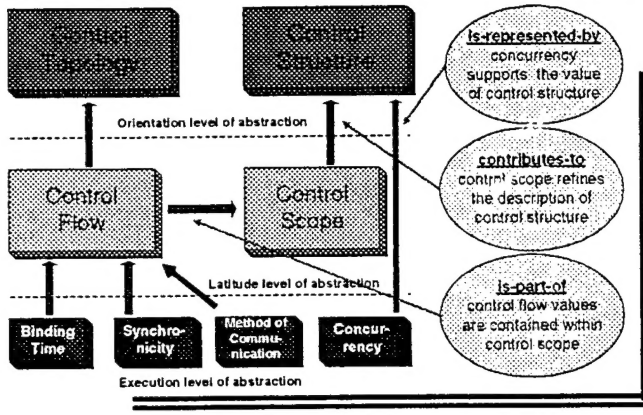
The PI was invited to participate in and present at the 1^{st} *Workshop on Evaluating Software Architectural Solutions* held in May 2000 at the University of California at Irvine.

The PI was invited to serve on a review panel for Software Engineering and Programming Languages Division of NSF and on the NSF Graduate Fellowship review panel.

The PI was invited to speak at the upcoming *Scuola Superiore G. Reiss Romoli Conference*, (SSGRR 2001).

The PI is organizing an invited session on Software Architecture Analysis for Component Interoperability at the 3^{rd} *International Conference on Information Reuse and Integration*, 2001.

Architectural Interaction Characteristics



Objective

Theoretical framework for control & data characteristics

Approach

Levels of abstraction to represent distinct viewpoints & strengths of characteristics

Semantic nets to establish direct and transitive connectivity

Accomplishments

- A principled and reusable methodology to qualify architectural characteristics.

  We define a minimal set of characteristics relevant to component interoperability analysis.

- The incorporation of composite application requirements in the form of characteristics.

  We determine how application requirements influence component interoperability.

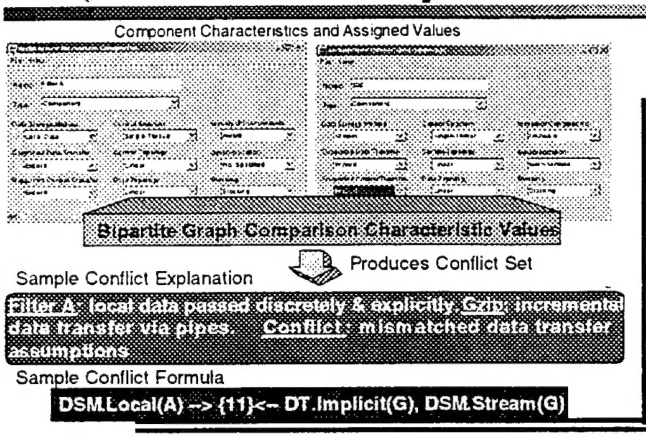
- The discovery of evolutionary qualities in architectural characteristics.

1

Rose Gamble, University of Tulsa

GRAPHIC REPRESENTING RECENT ACCOMPLISHMENT (1)

Comparison of Characteristics for Interoperability Assessment



Objective
To predict potential conflicts with component interaction

Approach
Cross comparison of characteristic values toward automation
Identify an enumerable conflict set
Establish a theory of problematic interactions

Accomplishments

- Utilization of established characteristics in the prediction of potential interoperability conflicts.

  We use a bipartite graph of the values from both components, extending conflict assessment beyond only similar characteristics.

- The determination of interoperability conflict patterns and their related characteristics.
- The development of an initial theory of problematic architecture interactions.

  The theory uses commutative rules and defines principles for conflict union & additivity.

Rose Gamble, University of Tulsa

GRAPHIC REPRESENTING RECENT ACCOMPLISHMENT (2)